A Reuse and Composition Protocol for Services

Dorothea Beringer
Stanford University
Gates Computer Science 4A
Stanford, CA 94305, USA
+1 (650) 725 0177
beringer@db.stanford.edu

Laurence Melloul
Stanford University
Gates Computer Science 4A
Stanford, CA 94305, USA
+1 (650) 723 0872
melloul@db.stanford.edu

Gio Wiederhold
Stanford University
Gates Computer Science 4A
Stanford, CA 94305, USA
+1 (650) 725 8363
gio@cs.stanford.edu

ABSTRACT

One important facet of software reuse is the reuse of autonomous and distributed computational services. Components or applications offering services stay at the provider's site, where they are developed, kept securely, operated, maintained, and updated. Instead of purchasing software components, the customer buys services by invoking methods provided by these remote applications over the Internet. Due to the autonomy of the services, this reuse model causes specific requirements that differ from those where one builds one's own system from components. These requirements have led to the definition of CPAM, a protocol for reusing remote and autonomous services. CPAM can be used on top of various distribution systems, and offers features like presetting of attributes, run-time estimation of costs and having several calls for setup, invocation and result extraction. The CPAM protocol has been successfully used within CHAIMS, a reuse environment that supports the generation of client applications based on CPAM.

Keywords

Internet-based Reuse, Interface Issues, Reuse Environments, Application Generators, Reuse Process

1. INTRODUCTION

Most models of reuse focus on systems assembled from components. Off-the-shelf components are bought from a supplier or acquired from a company-wide repository, their source code is copied into the application in which they are to be used, and they are compiled and linked with other components and glue code. If new versions of the components become available, it is up to the customer to purchase the new versions and to upgrade the applications using these components. The same paradigm holds for whole applications. A specific application is purchased and installed at the customer's site, and maybe integrated into other applications. In both cases, the provider sells actual code. Together with the code, responsibility for and control over the component is passed on to the customer. This includes the responsibility for installing, integrating, and maintaining the component, and the responsibility for providing resources and the control over these resources. In the case of components that are used within a larger program, the glue code composing these components is often written in the same programming language as the components, requiring intimate knowledge not only about the problem domain of the final application, but also of the programming language used as glue-code (e.g. C++, Java, Perl) and the interfaces of the components. The same is true for components in a distributed environment – in order to use remote components also knowledge of the distribution system and its programming is necessary (e.g. CORBA, DCE, RMI).

In the CHAIMS project our target is not composition and reuse of components or reuse by integrating applications, but composition and reuse of services (see figure 1). In contrast to reusing components, the programs providing services are not moved to the customer's site. The programs stay at the provider's site and various customers connect to the components or programs over the network, using the protocol CPAM on top of one or several of various possible distribution systems like CORBA, DCE, DCOM, RMI or plain TCP/IP. Control about the component and the resources needed stays with the provider, i.e. the provider is responsible for maintenance of the component as well as performance and availability of its services. We therefore speak of autonomous components. Of course, this reuse model is not applicable to small components like GUI-components or foundation classes, it is targeted at large, normally computation and/or data intensive components. We therefore call these components megamodules. Example for large components being offered in both ways, as remote services as well as locally installed applications are the various modules provided by Oracle Business OnLine [1]. In contrast to components at the client's site, remote components are used by several different clients and thus need to be laid out for collaborative use while providing privacy of data.

Reusing remote and autonomous services is not to be confused with classical outsourcing where customer specific applications are created and maintained at a provider's site for just one customer, and no other customers are using exactly the same applications and interfaces. CPAM is targeted at reuse – several customers use the same programs with different data over the same interfaces.

The focus of CHAIMS and CPAM are mainly on computational services, not just information services providing database accesses, as shown in figure 2. A traditional approach has been to have distributed information servers, and to have the processing at the client site (as shown in the left part of figure 2), if at all. In order to write and maintain such clients, domain knowledge as

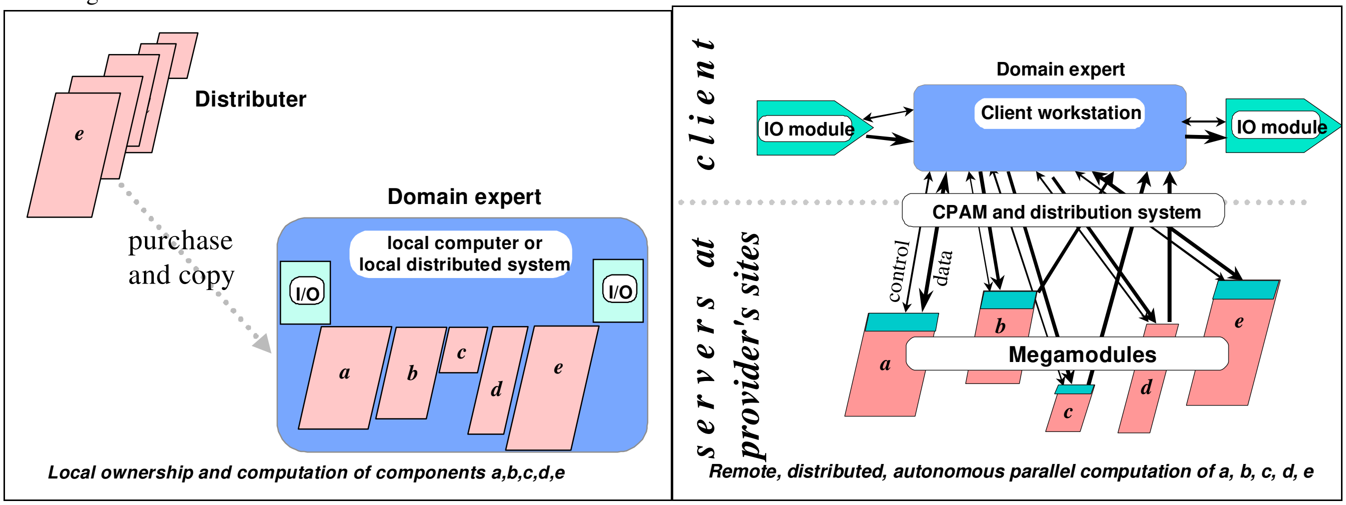

Figure 1: Reuse of local components versus reuse of services

well as detailed technical knowledge must be available at the client side. Often no reuse of these data processing components across various clients takes place. In our approach, shown in the right part of figure 2, servers not only provide access to some information stored in databases, the servers also perform the computation needed by the client. Data as well as computation resides on the server side, with the client focusing mainly just on composing these services. Both, the maintenance of any underlying databases as well as the maintenance of the modules and applications using these databases are under the responsibility of the service provider, and can be reused by several clients.

In the domain of web-information, all too often information is downloaded from an information or computational server, and put manually (cut and paste plus maybe some cumbersome conversions) into another program that performs further processing, e.g. a spreadsheet. A similar situation arises for many domains where there exist various services and programs for transforming and processing information, yet no integration of these services and programs exists, e.g. in the domain of genomics [2], [3], [4]. Genomic resources with integrated computation exist at many diverse sites. Today, these capabilities are used by an end-user invoking computations, cutting and pasting intermediate results into local workspaces, combining and editing results from multiple computations, and iterating manually to the desired result.

In order to automate the reuse and composition of computational services, a common protocol targeted at the reuse of distributed and autonomous services is necessary.

2. REUSING REMOTE, AUTONOMOUS SERVICES

Reusing services instead of components implies that the control over the component or megamodule remains at the provider's site. This has distinct advantages. The rules and processes encoded in a megamodule represent knowledge, yet this knowledge is subject

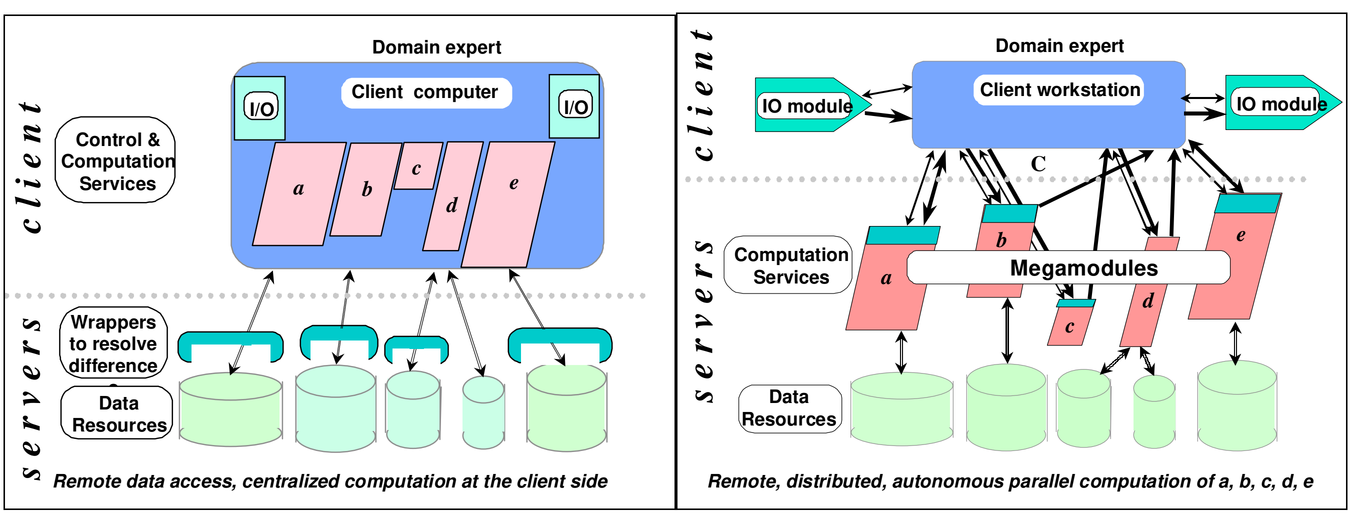

Figure 2: Simple database services versus computational services

to change, not only the data it operates on. Bringing the updated knowledge represented in algorithms and programs to the customer side, either by reusable components or as updated concepts and requirements that have to be integrated and implemented by the customer into its programs, is cumbersome without assistance. Also, in important large-scale cases no single person at the customer side can manage both, the maintenance of the programs and the exploitation of the results. By reusing services instead of creating or integrating purchased components and applications, the customer can narrow its focus on exploiting the results. The other tasks remain with the service providers.

Yet leaving the processing at the provider's site, and just reusing the services, also brings additional requirements:

- **Megamodules may be computation intensive and data intensive.** The duration of the computation cannot be neglected. It can range from a few seconds up to hours or even days. A simple request for information, e.g. flight information and calculation of flight options, is done within seconds. When we move to simulations, these can take hours. Yet because we deal with distributed computing, we can take advantage of the natural **parallelism** of many of the megamodules whenever we need results of more than one megamodule and these are not dependent on each other.
- **Megamodules are autonomous.** Megamodules are owned, operated and maintained by other people than the customers, and the customers of the megamodules have no direct influence on its resources and operation. There is no central controlling body directing the allocation of resources.
- **Megamodules are heterogeneous.** Megamodules are not only written in different languages, and run on different systems, also the middleware systems used to access the megamodules may be different. Some may be accessed over an ORB or a DCE system, others via RMI, DCOM or TCP/IP.

Due to the fact that megamodules are computation and data intensive, various cost factors have to be taken into account:

- **Time**: The time of a method's execution, i.e. the time from a method's invocation until the method can deliver the desired results. As megamodules can be computation intensive, this factor is important.
- **Fee**: Fee is the monetary cost of a service. The billing for internet services is still in its infancy, yet it will become more important when more and more services are offered to a wider public. Assuming autonomous megamodules, billing will be an integral part of using the methods of such megamodules. As fees can vary greatly, they can not be neglected when calculating the costs of using a specific method.
- **Data volume**: Megamodules can be data intensive. As a consequence, the amount of data that has to flow between the caller of a method (i.e. the customer using the service) and the megamodule cannot be neglected.

Because megamodules are autonomous, there is no central agency determining and controlling these cost factors. Yet a client might have to take these cost factors into account, and thus has to gain knowledge about them when invoking services. Time and fee can be estimated by the megamodule, be communicated to the client, and be directly used in any cost functions. For the third cost factor, data flow, the resulting amount of data can be estimated by the megamodule and be communicated to the client, yet the effective cost is not only determined by the amount of data but the time the data needs to be transferred. This time is client specific because it not only depends on the amount of data but also on the capacity of the connection, i.e. quality of connections, distance, and traffic volume.

3. CPAM, A PROTOCOL FOR REUSING SERVICES

3.1 Characteristics of CPAM

There are various application domains where reuse of services gets more and more important. Many web-based services providing processed information exist today, as weather services, airline ticket and book sales. Other potential services are simulation programs, design and construction programs, services for genomics [4] [5] and for manufacturing, business services [1], and many more are expected to come into existence. But there exist yet few protocols supporting an integrated vision and allowing easy reuse and composition into a larger system.

CPAM (CHAIMS Protocol for Autonomous Megamodules) is a protocol for accessing and using the methods offered by megamodules. We could also say that CPAM is a protocol for composing services.

CPAM has some special characteristics that are closely connected to the fact that CPAM addresses the composition and reuse of autonomous, mostly distributed and computation intensive services of megamodules, and not the composition and reuse of small local components, installed and executed within the same domain of control. These characteristics are: several calls for setup, method invocation and method extraction, the presetting of parameters, and the run-time estimation of costs (see figure 3). Having several calls allows to have a simple sequential client while exploiting the parallelism of methods from different megamodules, and provides us with an easy model for extracting pre-final results (e.g. from simulation services) and for extracting results from ongoing services (e.g. monitoring services). Only one protocol for different kinds of services is needed, and it includes an easy scheme for examining active method executions as well as aborting method executions. All these concepts become important when shifting from reusing local components or services within the same domain of control to reusing remote autonomous services.

CPAM has 9 primitives (see figure 3). In the current implementations of CPAM all the primitives are procedure calls from the client to the megamodules thus allowing simple sequential clients even when services are invoked in parallel.

3.2 Establishing a connection to a megamodule

The primitives SETUP and TERMINATEALL are used to setup the connection of a client to a megamodule, and to terminate this connection. Their only input parameter is *clientID*, an identification of the client reusing the services. In SETUP, this parameter tells the megamodule which client wants a connection, and allows the megamodule to setup the necessary internal data structures to handle all future calls of this client. With TERMINATEALL the client notifies the megamodule that it is no longer interested in any further services of the megamodule, and that the megamodule can kill any ongoing invocations and delete any client specific data like preset attributes.

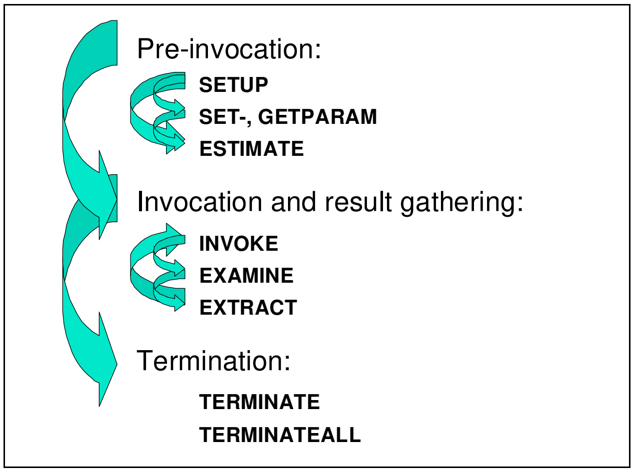

Figure 3: The 9 primitives of CPAM

3.3 Cost estimation

ESTIMATE allows a client to ask a megamodule for cost estimates for a specific method. The input parameters for ESTIMATE are the *clientID* of the calling client, a *method name*, and a *list of the names* of the cost factors requested. The method name tells the megamodule for which method an estimate is requested. The output parameter of ESTIMATE is a *name-value list* (name of the cost factor, value of the cost factor). The cost factors we have considered so far are execution time, execution fee, and data volume of the results. In the current version of CPAM, the ESTIMATE primitive does not allow to specify any invocation parameters. If these parameters influence the accuracy of the estimation, they can be preset by the SETPARAM primitive. Having cost estimation allows the client to choose among alternative services according to run-time criteria in case there exists more than one potentially suited megamodule for a specific task (different algorithms, different amount of resources, different fees, different availability). It also enables the client to schedule invocations of methods and to choose optimal execution paths; these are issues we will be working on in the future.

3.4 Executing megamodule methods

Methods are executed by the following four calls: INVOKE, EXAMINE, EXTRACT and TERMINATE. INVOKE starts the execution of a method, EXAMINE gives the status of the execution, EXTRACT returns desired results, and TERMINATE deletes the invocation.

INVOKE starts the execution of a method with a specific set of method attributes, also called method parameters in CPAM. Therefore one of the input parameters of INVOKE is a *name-value list* of attribute names and attribute values that have to be set specifically for this method execution, i.e. can neither be taken from default values nor from client specific presettings. Other input parameters of INVOKE are the *clientID*, used to notify the megamodule to which client it has to accredit this invocation, and the *name of the method* to be invoked. INVOKE has one output parameter, the *callID* which helps to identify this specific invocation in subsequent calls to the megamodule.

Because in CPAM it is always the client who initiates any communication with a megamodule and the megamodule has no possibility to inform the client of any event unless asked for, the client has to ask the megamodule periodically if results are ready or not. This is done with the call EXAMINE that takes as input parameter a *callID* to identify the invocation concerned and returns the *status of the invocation*. Besides DONE or NOT_DONE the status can also express to which degree an invocation is finished. This is needed when extracting preliminary results (e.g. in case of simulations), and can also be used for scheduling other invocations or aborting too slow invocations.

The results of an invocation are transferred to the client with the EXTRACT call. Its input parameter are the *callID* of the invocation concerned and a *list of the names* of result attributes to be extracted. This allows EXTRACT to do partial extraction of results whenever not all results are needed right away, or not all results are ready yet. EXTRACT returns a *list of attribute names and values.*

TERMINATE with the *callID* as input parameter is used to tell a megamodule that the client is no longer interested in a specific invocation. TERMINATE is necessary because for one invocation there may be zero, one or several extract calls until all results are extracted, and because a client is free to extract the same results several times or not to extract all results. Also, certain methods deliver ongoing results, e.g. in case of monitoring processes, so the client extracts results periodically. In cases where an invocation executes too slow, the client has gotten usable results already from another megamodule, or the client is no more interested in an invocation to produce any results due to some other circumstances (e.g. when starting lengthy method invocations in order to have the result when they are needed yet without knowing if the results will really be used), TERMINATE is also used to abort a method execution. For megamodules providing computation, abortion is no problem. In case of services that affect local status (e.g. reservation services), consistency of transactions becomes an issue. CPAM does not have itself any transaction related concepts. This is an application level issue and concerns the design of the services offered by a megamodule (e.g. offering a commit method), as well as the design of the applications using the protocol CPAM.

3.5 Presetting of attributes

The call SETPARAM is used to set default values for invocation attributes and global variables in a client-specific way. Its input parameters are the *clientID* and a *name-value list* containing the names and values of the attributes to be set. These attributes can be all of the attributes of the methods offered by the megamodule (method parameters), as well as global variables. Presetting of attributes is not only used for enabling pre-invocation estimates. It also prevents the costly retransfer of data whenever methods of the same megamodule are invoked several times by the same client with some of the attributes remaining unchanged. The call GETPARAM simply allows to investigate default values and client-specific presettings of attributes. GETPARAM takes as input parameter a *list of attribute names*, and returns a *list of* *names and values* of these attributes containing also descriptive name and type information (see below).

3.6 Client-specific versus invocation-specific primitives

The primitives INVOKE, ESTIMATE and EXTRACT are invocation specific. The attributes they set or the results they return are for/from one specific invocation, specified by the callID. Also TERMINATE is invocation specific, it just deletes the specified invocation.

The primitives SETPARAM and GETPARAM are client specific. The attributes they set or return are for one specific client, but are not linked to a specific invocation. A megamodule may have several client-specific settings of the same attributes, because it may have several clients using it at the same time. Also SETUP and TERMINATEALL are client specific, TERMINATEALL kills all invocations of a specific client.

The primitive ESTIMATE is client and method specific, but not invocation specific. The estimates it returns are for the specified method, and for the preset (or default) attribute settings of the client asking for the estimates.

3.7 Attribute names and values

CPAM does not require that all possible attributes appear in the instantiation of a call, or that they appear in a specific order. Therefore attribute names are needed for the identification of attribute values. When routing attribute values from one megamodule on to another megamodule, something done quite frequently when using services of various megamodules, a common data format must be used. The use of CPAM is not limited to a specific distribution system. Yet this inhibits the use of a distribution system specific data format like e.g. the CORBA IDL type system. Furthermore we do not want to restrict megamodule services to an inflexible format and content of data they provide and accept, or to require them to publish these formats and stick to them forever. We rather want to give megamodules and suites of megamodules the possibility to adapt and change content and type of attribute values, just as they can change algorithms, rules and data bases of the processing they provide. Therefore attribute values do not only contain the actual data, but also description of the data (descriptive name) and type information, thus giving the megamodules the necessary flexibility they need due to their autonomy. Attribute values in CPAM are so called **gentypes** (see figure 4). A gentype is a recursive data structure of data elements with each data element consisting of a descriptive name, type information, and a value. The value is either again a gentype, or, if the data element is of a simple type like bit-map, array of bytes, datetime, string, real etc., the value holds the effective value of that data element. Examples for attributes that only contain a simple gentype are the attributes "fee" and "time" used in the primitive ESTIMATE; "fee" is an integer whereas "time" is of type datetime. The descriptive name can be arbitrary text describing the data element and is not linked at all to the attribute name. In order to facilitate the use of CPAM across heterogeneous distribution systems, gentypes use the type system of ASN.1 and are encoded with BER [6].

An alternative to using gentypes and ASN.1/BER is the emerging standard XML [7], with megamodule suite specific DTDs. This possibility will become especially interesting as soon as more tools will be available for an easy integration of XML into systems using CPAM.

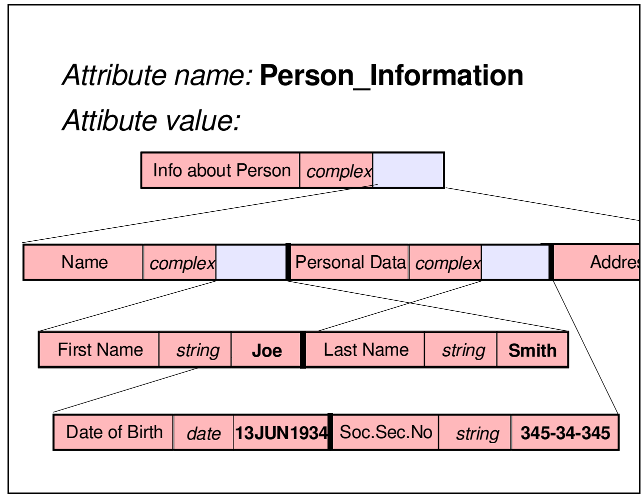

Figure 4: Attribute values

3.8 Order of primitives

CPAM primitives cannot be called in any arbitrary order, the following preconditions have to be fulfilled:

- All primitives apart from SETUP: Connection to the megamodule must have been established by SETUP and may not yet have been terminated by TERMINATEALL.
- EXAMINE, EXTRACT, TERMINATE: The invocation to be examined, extracted or terminated must have been invoked by INVOKE, and may not yet have been terminated by TERMINATE.

Apart from these two constraints, the order of primitives is free. The client can estimate the cost of a method before, during or after its invocation. Results may be extracted from a method without prior examination of the status of the method. Yet the client carries the risk of incomplete or wrong results unless it knows by some other mean that the results it wants are ready. This is the case for services that are ongoing monitoring processes and that state, e.g. in a repository, that after an invocation valid results are always available for extraction. An invocation may be terminated without any prior result extraction, e.g. when methods like print jobs do not compute any results for the client.

3.9 Heterogeneity

We assume that megamodules are heterogeneous concerning programming language and platform. Yet heterogeneity also concerns the distribution protocol used to access these megamodules (CORBA, RMI, DCOM, DCE, TCP/IP). CPAM does not define any protocol for transport and interconnectivity. Instead, it uses one or several of the existing distribution protocols for creating a connection to a megamodule and for transporting the various primitives of CPAM. Therefore, above specification of CPAM may be implemented on top of various distribution systems. So far we have defined CPAM protocols for CORBA, RMI, DCE, TCP/IP and for local C++ and Java (see http://www-db.stanford.edu/CHAIMS/Doc/Details). Our current implementations use two different versions of ORBs, as well as RMI. By having several CPAM implementations for different distribution protocols and by layering CPAM on top of these distribution protocols, we avoid the limitation to one specific distribution system.

CPAM is not the only protocol that splits up method invocation and result extraction. This can also be found in the DII (Dynamic Invocation Interface) of CORBA and in the proposed SWAP protocol (Simple Workflow Access Protocol) [8]. Yet the detailed mechanisms for progress testing and result extraction differ. Due to their different focus and context they also do not provide any mechanism for pre-invocation estimation.

4. THE CHAIMS ENVIRONMENT

The use of the protocol CPAM is not restricted to a specific environment. So far we have investigated its use in two different settings: with a SQL-based language as front-end that composes the megamodule methods in a way similar to data [Burback, personal communication], and within the CHAIMS system (Compiling High-level Access Interfaces for Multi-site Software) [9] with the composition language CLAM [10] as front-end.

As shown in figure 5, the main components of the CHAIMS system are the repository, the CHAIMS compiler, and the wrapper templates [11]. The **repository** contains a description of all megamodules, their methods, their attributes, the underlying distribution protocol used by the megamodule, and its location. All the valid megamodule, method and attribute names are posted in the repository. The repository is the only information flow necessary between those persons providing megamodules and those using their services. The exchange of repository information is simple because the repository is in readable text format. For ease of use we also provide a user-friendly graphical front-end to the repository. The **CHAIMS compiler** compiles a megaprogram written in the composition language CLAM into a client side run-time (CSRT), inclusive the generation and compilation of all necessary stubs for various distribution systems, based on the information found in the repository and the definition of the CPAM protocol. The **wrapper templates** are provided as part of the CHAIMS system in order to facilitate the wrapping of legacy modules into CPAM conformant megamodules.

The composition language CLAM hides from the megaprogrammer, who is a domain expert reusing services of megamodules, technical details like the use of complex programming languages and the programming of distribution systems. Because the megamodules are autonomous and they are created, installed and maintained from people other than the ones using their services, the megaprogrammer does not have to know anything about their technical internals. Thus, it is a logical consequence not to require any technical knowledge from the megaprogrammer on the client side, in order to facilitate the composition and reuse of services for non-technical domain experts, as it is done e.g. in CHAIMS. This can be compared to what is seen today in the use of database management systems, where the SQL programmers are quite distinct from the programmers who work at the DBMS provider, and it is unlikely that they have ever met. We hence assume that these two roles are occupied by different persons with differing skills and objectives.

If CPAM is used in the context of CHAIMS, then the types of attributes, unless restricted by the repository to one of the basic CHAIMS types, are opaque to CPAM as well as CLAM. Neither CPAM nor the user of CPAM have to know it, because within CHAIMS they just route data from one megamodule to another one. CLAM, as a pure composition language, and CHAIMS as a system with a clear separation between data-view, composition-view and transportation-view, leave the investigation and interpretation of attribute values to the megamodules, with the exception of the results of the estimate primitive. Therefore, apart from the attributes "fee", "datavolume" and "time" of the estimate primitive, the attribute types can be defined simply as opaque in the repository.

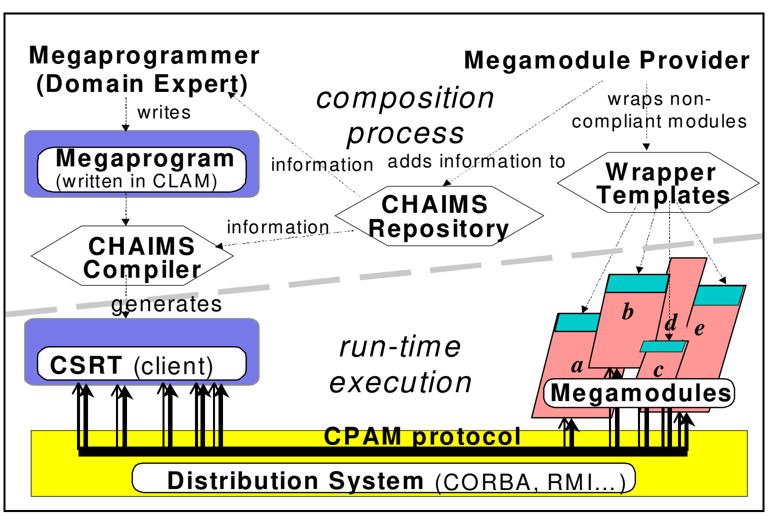

Figure 5: The CHAIMS system

There exist other systems for composing distributed components, e.g., Hadas [12] and Regis [13]. But in contrast to CHAIMS, these systems do not assume the components or services they compose to be all of the following: distributed over sites of different organizations, autonomous, heterogeneous also concerning the distribution system, computation intensive. They therefore do not have a protocol like CPAM especially targeted at these issues.

5. CONCLUSIONS

The reuse and composition system CHAIMS as well as the protocol CPAM, a protocol for reusing autonomous megamodules, are based on a megaprogramming paradigm. Megaprogramming refers to the creation of large-scale programs through a process of composition of autonomous programs and modules [14]. In a megaprogramming approach, composers are willing to give up control for the benefit of expert maintenance at the source sites in a collaborative setting [15]. Megaprogramming distinguishes itself from database integration by composing knowledge embedded in programs, rather than being limited to declarative knowledge applied to databases. Database functionality can be incorporated into megaprogramming through server programs that execute SQL SELECT statements, but these languages - focusing on a single verb - are known to have inherently limited computational capabilities [16].

Megaprogramming can also be viewed as large-scale object-oriented (OO) technology. OO increases the procedural capabilities of distributed objects [17], but is restricted in practice to single protocols and coherent libraries [18]. In contrast of having reuse by purchasing, copying and integrating code, or of having distributed objects within one company under one central control, CPAM, as a specific example of a protocol used for megaprogramming, scales the object-oriented paradigm to autonomous service objects.

CPAM has been implemented at Stanford University as part of the CHAIMS project. Case studies include a logistics example ("find the best route from city A to city B under certain circumstances") using several megamodules for the various parts of the computation, and an aircraft design example with megamodules for the computation of the structure, the control elements and the static of an aircraft wing. CPAM is just one important piece in the process of reusing autonomous services. Just as important is a reuse environment in which the advantages of a protocol like CPAM can be fully exploited. In the current and future focus of our research we are improving the CHAIMS environment with composition wizard, wrapper wizard and repository browser, and we are going to integrate automatic invocation scheduling into the CHAIMS compiler and the generated client. Based on the primitives of CPAM, especially the ESTIMATE and the partial EXTRACT primitive, the goal of the automatic invocation scheduling will be to optimize overall costs, i.e. overall fees as well as time. Other future research issues in the CHAIMS project are the integration of security and error-handling, preferably by reusing features of the underlying protocols. As new technologies for inter organisational communication evolve, it will also be interesting to see how the main ideas of the CPAM protocol can be mapped into these systems.

The CHAIMS project is supported by DARPA order D884 under the ISO EDCS program, with the Rome Laboratories being the managing agent, and also by Siemens Corporate Research, Princeton, NJ. We also thank the referees of SSR'99 for their valuable input, and the various master and PhD students that have contributed to the CHAIMS project.

6. REFERENCES

[1] "Oracle Business OnLine, Removing Barriers to Enterprise Applications Adoption", see http://www.oracle.com/businessonline/

[2] B. Altman, N. F. Abernethy, R. O. Chen: "Standardized Representations of the Literature: Combining Diverse Sources of Ribosomal Data." Proceedings of the Fifth International Conference on Intelligent Systems in Molecular Biology, Halikidiki, Greece, 1997, AAAI Press, Menlo Park, p.15-24.

[3] S.B. Davidson, C. Overton and P. Buneman: "Challenges in Integrating Biological Data Sources"; Computational Biology 2, 1995, pp 557-572.

[4] J. H. Gennari, H. Cheng, R. B. Altman, & M. A. Musen: "Reuse, CORBA, and Knowledge-Based Systems"; Int. J. Human-Computer Sys., in press, 1998.Content-Length: 1773

[5] David Searls; "Biowidgets"; Computational Methods in Molecular Biology, Elsevier Science, 1998.

[6] "Information Processing -- Open Systems Interconnection -- Specification of Abstract Syntax Notation One" and "Specification of Basic Encoding Rules for Abstract Syntax Notation One", International Organization for Standardization and International Electrotechnical Committee, International Standards 8824 and 8825, 1987.

[7] "Extensible Markup Language (XML), 1.0", Recommendation of the World Wide Web Consortium, February 1998.

[8] Keith Swenson; "Simple Workflow Access Protocol (SWAP)", Internet-Draft submitted to WfMC (Workflow Management Coalition), available at http://www.ics.uci.edu/pub/ietf/swap/

[9] L. Perrochon, G. Wiederhold, R. Burback; "A Compiler for Composition: CHAIMS"; Fifth International Symposium on Assessment of Software Tools and Technologies (SAST'97), Pittsburgh, June 3-5, 1997.

[10] N. Sample, D. Beringer, L. Melloul, G. Wiederhold, "The coordination language CLAM", Coordination'99, Amsterdam, Netherlands, April 1999.

[11] D. Beringer, C. Tornabene, P. Jain, G. Wiederhold: "A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules"; DEXA International Workshop on Large-Scale Software Composition, Vienna Austria , August 1998.

[12] I. Ben-Shaul, et.al.: "HADAS: A Network-Centric Framework for Interoperability Programming", International Journal of Cooperative Information Systems, 1997

[13] J. Magee, N. Dulay, J. Kramer; “Regis: A Constructive Development Environment for Distributed Programs”, IEE/IOP/BCS Distributed Systems Engineering, 1(5): 304-312, Sept 1994.

[14] B. Boehm and B. Scherlis: “Megaprogramming”; Proc. DARPA Software Technology Conference 1992, Los Angeles CA, April 28-30, Meridien Corp., Arlington VA 1992.

[15] Gio Wiederhold, P. Wegner and S. Ceri: “Towards Megaprogramming: A Paradigm for Component-Based Programming”; Communications of the ACM, 1992(11): p.89-99.

[16] J. D. Ullman: “Principles of Database and Knowledge-Base Systems; Volume 1: Classical Database Systems”, Computer Science Press, 1988.

[17] Grady Booch: “Object-Oriented Design with Applications, 2nd Ed.”; Benjamin-Cummins, 1994.

[18] M. P. Atkinson, V. Benzaken, D. Maier (eds.): "Persistent Object Systems"; Springer-Verlag and British Computer Society, 1995, Workshops in Computing Series, ISBN 3-540-19912-8.